INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

SIMULATING GAMES USING OBJECT-ORIENTED METHODOLOGY

HONGLANG LI

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 1998
© HONGLANG LI, 1998

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-39988-5

Canada

Abstract

Simulating Games Using Object-Oriented Methodology

HONGLANG LI

In this report, we present a Bridge simulator and we discuss object-oriented analysis, design and programming. The design phase uses automated support to illustrate how we apply the concepts of object-oriented methodology to develop software — a Bridge simulator. The implementation of the Bridge simulator demonstrates the programming process by using an object-oriented language(C++). Important features of the Bridge simulator are the use of the object-oriented paradigm for design and the use of the X Window/Motif toolkits to construct a user interface for simulating the bidding and the playing of the game of Bridge. We conclude with the results of the Bridge simulator, discuss a research on computer Bridge and suggest avenues for further directions in which the project could be extended.

Acknowledgments

I wish to express my sincere gratitude to my supervisor, Dr. Peter Grogono, for all his enthusiastic support, careful supervision, and consistent guidance during the development of this major report. I also wish to thank Dr. Clement Lam, who kindly took the time to review the report.

Furthermore, I would like to appreciate my friends Kevin Theobald, Baoshuo Chen, Louis Harvey for their ideas, suggestions and help during the period of the project.

Finally, I wish to thank my parents for all their encouragement and support.

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

1.1 Aim of the Project 1

1.2 Motivation 1

1.3 Background 2

1.4 The Structure of the Report 2

2 The Game of Bridge 3

2.1 The Basic Rules 3

2.1.1 Bidding 3

2.1.2 Play 4

3 Object-Oriented Concepts and X Window/Motif 5

3.1 Object-Oriented Technology 5

3.2 Object-Oriented Methods 8

3.2.1 CRC: Classes/Responsibilities/Collaborators 8

3.2.2 Responsibility-Driven Design 9

3.3 X Window System and Motif 10

3.3.1 The X Window/Motif Architecture 10

4 Design 12

4.1 Identify the Classes 12

4.2 Classes Design 12

4.2.1 Class Card Specification 13

4.2.2 Collaboration Graph 20
4.3 Design Bridge User Interface 20
4.3.1 Basic Considerations 20

5 Implementation **24**
5.1 Implementation Issues 24
5.1.1 Implementation Languages 24
5.1.2 User Interface Components 25
5.1.3 Problems in C++ with Motif 26
5.1.4 Programming Process 26
5.2 System Structures 27
5.3 Features of Windows 29
5.4 Overview of Code 31

6 Results **35**

7 Conclusion **41**
7.1 Bridge Simulator System 41
7.1.1 Experience on Object-Oriented Programming 41
7.1.2 The Advantages of C++ 42
7.2 Computer Bridge 43
7.3 Further Work 44

Bibliography **45**

Appendix **48**

A card.h **48**

B deck.h **49**

C player.h **50**

D dealer.h **51**

E bidboard.h **52**

F command.h **53**

G message.h 54

H handboard.h 55

I bridge.h 57

List of Figures

1 A simple collaboration graph 9
2 The architecture of the X Window system 11
3 Card class card 13
4 Deck class card 14
5 Player class card 15
6 Dealer class card 16
7 BidBoard class card 16
8 Message class card 17
9 Command class card 18
10 HandBoard class card 19
11 Bridge class card 19
12 The Bridge collaboration graph 21
13 Bridge simulator user interface layout 22
14 Class structure diagram 27
15 Motif widget structure diagram 28
16 System callback structure 30
17 Example 1 38
18 Example 2 40

List of Tables

1 Abbreviation for bidding board 36
2 Abbreviation for hands' board 36

Chapter 1

Introduction

In this chapter, we describe the project that we selected and why we chose this topic. We also introduce the background of the project and describe the structure of this report.

1.1 Aim of the Project

This project is about game simulation. Our particular example will be the game of Bridge. The aim of this project is to present an object-oriented approach to simulating Bridge using C++ and the Motif user interface toolkits. The original intention is to design and implement a Bridge simulator as realistically as possible. Therefore a user interface is constructed so that people can use it as a tool to learn and to play the game of Bridge. Also, by using object-oriented methodology, we plan to generate some classes which can be basic modular classes for general card games so that people can use these classes for designing any card game.

1.2 Motivation

Games are very interesting topics which always attract people doing research or programming in computer literature because:

- They provide a good model to establish data structures and algorithms;
- They combine knowledge of artificial intelligence and gaming strategies;

- They require a rich and complex computer user interface.

Designing a Bridge simulator is an interesting project because Bridge is a game that requires considerable intelligence. Computer Bridge simulator is an important element in the study of artificial intelligence and pattern recognition; an important model in the establishment of data structures and algorithms; and an important application in the development of a computer user interface with object-oriented methodology.

1.3 Background

Bridge, as one of the best card games, is enjoyed by millions of participants and is commonly played around the world. It is not surprising that Bridge is highly appealing because it combines many fascinating features. Some of these are:

- Bridge is a game of skill. It is sufficiently demanding to provide a challenge to people: it requires such abilities as memory, reasoning, judgement, and planning.
- Bridge is a team game. The behaviour patterns of your opponents is an intriguing aspect of Bridge. The interplay of skill and chance is one of the most appealing features of Bridge.
- In Bridge, exact situations are virtually never duplicated. This factor increases the complexity of Bridge.

1.4 The Structure of the Report

Chapter 1 is an introduction for this report. It describes briefly the motivation and background of the project. Chapter 2 presents an introduction to the game of Bridge. Chapter 3 describes the object-oriented technology and methods, combining a CRC card method with a Responsibility-Driven Design method; it also describes the X Window/Motif toolkits for building up a computer user interface. The design of the Bridge simulator is presented in chapter 4. It takes advantage of the methods described in the chapter 3. Chapter 5 gives the details of the implementation of the Bridge simulator. Chapter 6 presents the result of the Bridge simulator with examples. Chapter 7 concludes this project and summarises the object-oriented design and programming. It also suggests further work for the project.

Chapter 2

The Game of Bridge

Here. we present a brief introduction to the game of Bridge. If readers are interested in further details. they could refer to the books on the subject [9] and [10].

2.1 The Basic Rules

Bridge is a card game played with a deck containing 52 cards. comprised of 4 suits (Spades. Hearts. Diamonds. and Clubs) each containing the 13 cards Ace. King. Queen. Jack. 10. .... 2 (we will sometimes abbreviate the first four of these to A. K. Q. and J). The game begins with a random shuffling the deck. and the cards are then dealt to four players. traditionally named North. South. East and West. Each player receives 13 cards. These players form two teams: North/South against East/West. The game contains the two stages of bidding (or auction) and play.

2.1.1 Bidding

The bidding stage determines which team wins a *contract* to make a certain number of tricks. During the bidding stage. the players take turns to make "bids". A bid is (n. s) when n is a number from 1 to 7 and s is a suit. Each bid must be higher than the previous one according to the convention that (n1. s1) > (n2. s2) iff either n1>n2. or n1 = n2 and s1 > s2 in the ranking *NoTrump* > *Spade*> *Heart* > *Diamond* > *Club*. When no player wishes to bid further (i.e. all players pass). the highest bid is regarded as an offer to play a *contract*. in effect a bet that the player who has made the final bid (n, s) can. together with his partner. take at least 6+n tricks with suits

as trumps. It is also possible, during the bidding, to *pass* a bid by any player, to *double* a bid by the other side or to *redouble* the opponent's double. But doubling affects the score, not the number of tricks to be made.

After the Bidding stage, one member of the winning team becomes the *declarer* and the other is the *dummy*. The other team are the *defenders*. The defender to the left of the declarer leads the first card, and the dummy's cards are then laid face-up on the table from where they are played by the declarer, along with his own cards.

2.1.2 Play

Card play starts when the defender to the left of the declarer lays a card on the table, which all the other players then cover in turn (in a clockwise direction) with a card from their own hand. Each round of four cards is called a *trick*, and the winner of one trick becomes the first person to play a card on the succeeding round. The basic rules that govern card play are:

- The first player in a trick can freely choose which card to play from all those present in his hand.

- Subsequent players must follow suit by playing a card of the same suit as the one that started the trick if they hold such a card. If they do not (i.e. they are *void* in the suit), they can make a free choice from among the remaining cards in their hand.

- The winner of the trick is the player who plays the highest card (ranked by A > K > Q > J> 10 > ... > 2) of the suit led. The only exception to this is when there is a suit declared as the *trump suit*. If any trump cards are played, then the player playing the highest trump card is the winner. Playing a trump when you cannot follow suit is known as *ruffing*.

Chapter 3

Object-Oriented Concepts and X Window/Motif

We have used object-oriented techniques and the X Window/Motif toolkits to design and implement a Bridge simulator. Before proceeding with the design of the Bridge simulator, it will be helpful to introduce the basic concepts of object-oriented technology and methods which we applied in our design. We also give a brief introduction to the X Window system and Motif toolkits.

3.1 Object-Oriented Technology

Object-oriented technology is more than a way of programming; it is a way of thinking abstractly about a problem using real world concepts, rather than computer concepts. It provides a practical and productive way to develop high quality software for many applications. The term **object-oriented** means that we organise software as a collection of discrete objects that incorporate both data structure and behaviour. This is in contrast to conventional programming in which data structure and behaviour are only loosely connected [11].

As the name object-oriented implies, objects are key to understanding object-oriented technology. An **object** is characterised by a number of operations and a state which remembers the effect of these operations [7]. A software object has two

characteristics: state and behaviour. It maintains its state in variables and implements its behaviour with methods.

Object-oriented analysis, design and programming methodologies work together to produce a combination that better model their problem-domains than similar systems produced by structured techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust and promote greater design and code reuse.

Object-Oriented Analysis (OOA) is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain [1]. A **class** represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operations and for their information structure [7].

OOA aims at understanding the system to be developed and building a logical model of the system. This model is based on natural objects found in the problem domain. The objects hold data and have behaviour in terms of which the entire system behaviour can be expressed. OOA contains the following activities [7]:

- Finding the objects,
- Organising the objects,
- Describing how the objects interact,
- Defining the operations of the objects,
- Defining the objects internally.

Object-Oriented Design (OOD) is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design; specifically, this notation includes class diagrams, object diagrams, module diagrams, and process diagrams [1].

Object-oriented design turns from modelling the problem domain towards modelling the implementation domain. Because object-oriented analysis and design use the same notations, it is natural to have design and implementation running in parallel and iteratively. As Booch [1] points out: The boundaries between analysis and design are fuzzy, although the focus of each is quite distinct. In analysis, we seek to model the world by discovering the classes and objects that form the vocabulary of the problem domain, and in design, we invent the abstractions and mechanisms that provide the behaviour that this model requires.

Object-Oriented Programming (OOP) is a method of implementation in which programs are organised as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships [1].

The major strength of OOP is that it encourages the reuse of code and that it is usually easier to understand and maintain than other types of programming. The key to programming in an object-oriented programming language are the classes. In the class the programmer defines the variables and the operations associated with the class and instances of the class. From these classes, instances are dynamically created during execution of the program. An **instance** is an object created from a class. The class describes the (behaviour and information) structure of the instance, while the current state of the instance is determined by the operations performed on the instance [7]. An object-oriented language must support the following: encapsulated objects, the class and instance concepts, inheritance, and polymorphism.

How are OOA, OOD, and OOP related? Basically, the products of object-oriented analysis serve as the models from which we may start an object-oriented design; the products of an object-oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods [1].

3.2 Object-Oriented Methods

There are many object-oriented development methods that have been proposed for designing applications, but few are widely accepted and applied. Some proposed approaches emphasise comprehensive, theoretical techniques for developing object-oriented systems, while others are based on rapid prototyping in an interpretive environment. Even though the methods are different, the goal of each of them is the same, that is to allow the designer to start from a problem and develop a set of classes that can be used to implement a solution.

Based on our application, we introduce two well-known object-oriented design methods and use the methods to design and implement the Bridge simulator.

3.2.1 CRC: Classes/Responsibilities/Collaborators

Kent Beck and Ward Cunningham [3] have developed a method for designing object-oriented systems. This method is known as CRC cards, which characterise objects by class name, responsibilities, and collaborators. The method uses abstract principles that apply to any object-oriented system, regardless of the implementation language. CRC helps the designers identify objects and relationships between objects, while building a common understanding of a system.

The method concentrates on identifying the key responsibilities of each class and also identifying relationships between classes. Other classes that are involved in a responsibility are known as *collaborators*. Classes may depend on collaborators to provide information or to perform operations.

The method presents a CRC card which stands for a Class, its Responsibilities, and its Collaborating classes. The class name is written across the top of the card. Each card contains a list of the classes' responsibilities and a list of other classes that collaborate with the class. A responsibility is a short verb phrase that identifies a particular problem solved by an object. The responsibility indicates what the object does but not how the task is accomplished.

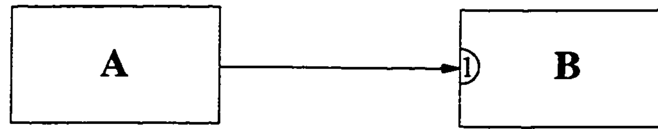

Figure 1: A simple collaboration graph

An important characteristic of the CRC approach is that the distinction between objects and classes is blurred. At early stages of a design, it is not always necessary to distinguish between instances of a class and the class itself. However, the responsibilities of a CRC card should represent the behaviour of the objects which is defined by the classes.

3.2.2 Responsibility-Driven Design

The book *Designing Object-Oriented Software* [13] describes a design approach that expands the simple ideas found in CRC into a more comprehensive and detailed process. CRC cards simply list all collaborators along the right side of the class card. Wirfs-Brock et al. introduce the idea that collaborators should be closely associated with a specific responsibility. Collaboration can be mutual relationships between two classes or they can be unidirectional.

Wirfs-Brock introduces another technique, along with a supporting notation, which is very helpful for developing and describing designs. This notation is known as a *collaboration graph*. A collaboration graph shows the various relationships between the collaborators identified on the class cards. Collaboration graphs can help programmers visualise the overall architecture of an object-oriented system. Figure 1 shows a typical collaboration graph containing the classes A and B. The rectangular boxes represent individual objects (or classes). The semicircle along the side of the box on the right represents a certain responsibility handled by the class. A vector connecting two classes represents a collaboration between those classes. The arrow points to a relevant responsibility handled by the collaboration class.

3.3 X Window System and Motif

The X Window system (often known simply as X) is an industry standard window system that provides a portable base for applications with graphical user interfaces. Motif is a high-level user interface toolkits that makes it easier to write applications that use the X Window system [14].

3.3.1 The X Window/Motif Architecture

Motif applications use three distinct libraries: **Xlib, Xt** Intrinsics. and **Motif** widgets.

Xlib provides the principal C language interface to the services supplied by the X server. This library encapsulates the mechanisms for connecting to the X server, sending requests to the server. and receiving events back from the server. The functions in Xlib allow applications to create windows. move and resize windows. draw text and graphics to windows. and so on [14].

Xt Intrinsics was also written in C. It defines the abstract base classes on which user interface components can be built. It also defines the external protocol used by applications to interact with all components based on Xt. The user interface components supported by Xt are called *widgets*. such as buttons. scrollbars. and menus. A widget consists of a structure that contains data and support functions that operate on those data. Each widget has an X window associated with it. in which the widget displays itself [14].

Motif is a standard user interface toolkits that consists of several parts [14]:

- The Motif widget set is based on the Xt Intrinsics.
- The Motif Style Guide contains rules and recommendations for achieving visual and behavioral consistency with other Motif applications.
- The Motif window manager. mwm. is an *Inter Client Communications Conventions Manual* (ICCCM) compliant window manager whose appearance and behaviour complement the Motif widget set.

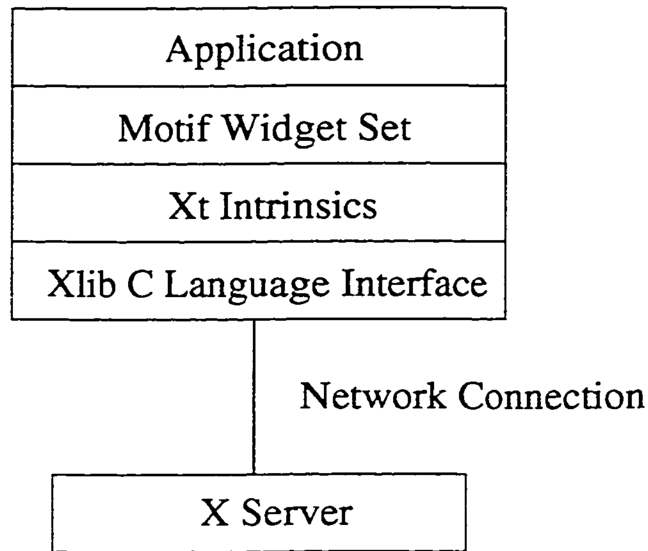

Figure 2: The architecture of the X Window system

- User Interface Language (UIL) offers an alternative to the C-language interface for Motif-based applications.

Figure 2 shows how these libraries relate to one another. The X server is a process that normally runs on a local machine and communicates with clients across a network protocol. The Xlib layer handles the network traffic on the client side and presents a low-level C language interface to the facilities of the X server. The Xt Intrinsics library is built on top of Xlib and hides many of Xlib's lower-level details. Motif is built primarily on the Xt layer but occasionally calls Xlib functions directly [14]. These three libraries provide a powerful and flexible toolkits to create a user interface.

Chapter 4

Design

Our design notation is based on the CRC and Responsibility-Driven approaches described in the previous chapter. We first identify classes of the Bridge simulator system, describe CRC cards for each class, then draw the collaboration graph to show the relationship among the classes of the system.

4.1 Identify the Classes

The key for designing a Bridge simulator using object-oriented methods is to identify the objects and classes of objects in the system. As described in chapter 2, we find the nouns such as card, deck, player, and dealer that would represent the objects in the Bridge game. For the Bridge simulator, we design a user interface which is represented by the objects such as message, command, bidding, hand, and bridge. Therefore, two categories of classes are defined in the system. One category of class is associated with non-user interface which contains the classes: Card, Deck, Player, and Dealer. The other is associated with the user interface and contains the classes: Message, Command, BidBoard, HandBoard, and Bridge.

4.2 Classes Design

In this section, we will take advantage of the method which has been described in section 3.2 to specify each class in the Bridge simulator.

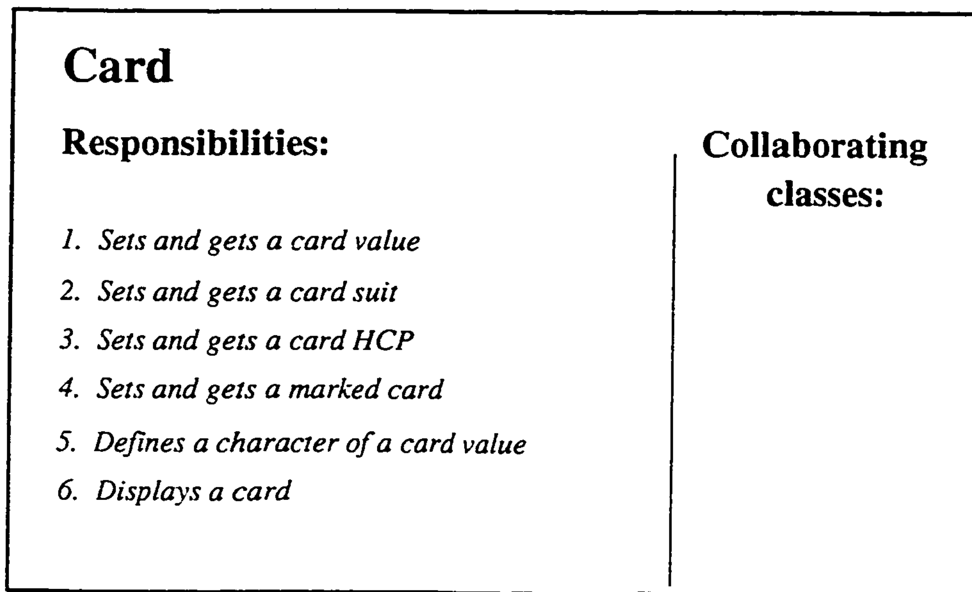

Figure 3: Card class card

4.2.1 Class Card Specification

Class *Card*
A card is an essential element of Bridge and it contains attributes of value and suit. The cards in each suit rank from highest to lowest: A, K, Q, J, ..., 2. The first four cards are the most powerful cards in Bridge. There are certain points called High Card Points (HCP) associated with them. The HCP of A is 4, K is 3, Q is 2, and J is 1. The Card class describes a single playing card. It is responsible for a set of operations for a single card. The Card class card is shown in Figure 3.

Class *Deck*
Bridge is played with a standard collection of 52 cards called a *deck*. The deck is divided into four suits which have specific ranks. Spades have the highest rank, then Hearts, Diamonds, and Clubs; Clubs have the lowest rank. The two highest-ranking suits, Spades and Hearts, are called the major suits; the two lowest-ranking suits, Diamonds and Clubs, are called the minor suits. The Deck class consists of an ordered collection of cards, along with the responsibility of setting the HCP of the topmost four cards.

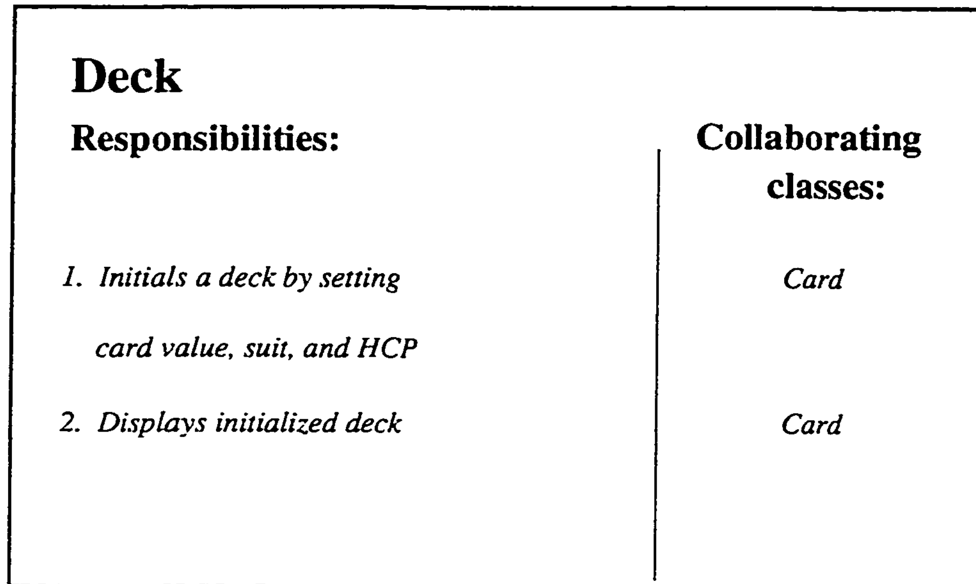

Figure 4: Deck class card

The object of a Deck class is responsible for initialising itself in order and displaying a deck if it is necessary. Carrying out initialisation involves setting the value, suit and HCP of the 52 cards, which is maintained by the class Card. Therefore, the class Card is a collaborator. The Deck class card is shown in Figure 4.

Class *Player*
Bridge is a game for four players. Two of them sitting opposite each other are partners. It is traditional to refer to the players according to their position at the table as North, East, South and West; so North and South are partners playing against East and West. A player is an important object of the Bridge game. It has therefore been defined to be a basic class of the system.

A *hand* represents a player and it holds 13 cards randomly chosen from a deck. A player needs to rearrange the cards in the hand, which includes arranging card suit of the hand, in the order Spade, Heart, Diamond, and Club; sorting the hand's card value in an descending order for each suit; calculating the hand's HCP; defining the characters of the hand's card value. Thus, the class Player needs to collaborate with the class Card to perform its responsibilities. Figure 5 shows the Player class card.

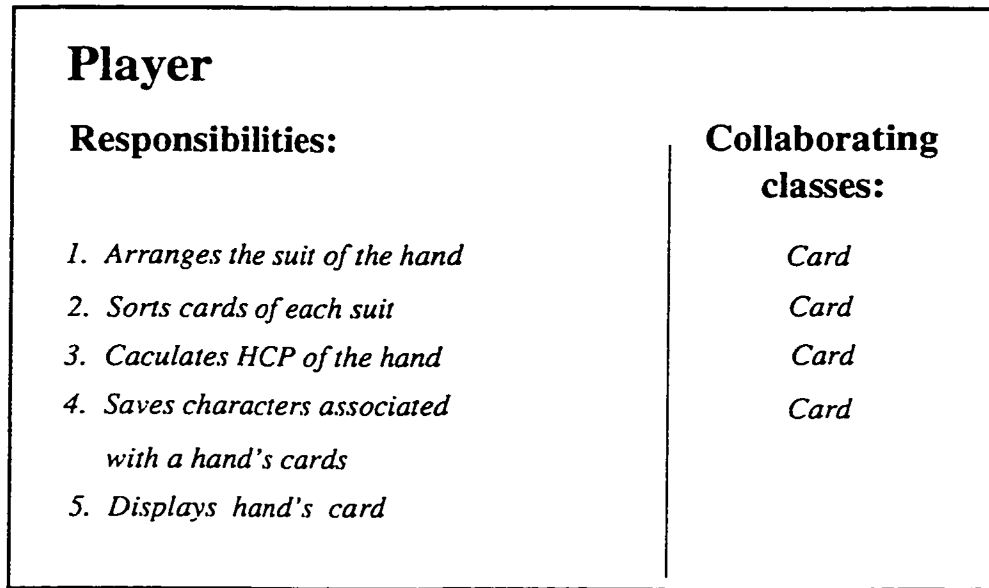

Figure 5: Player class card

Class *Dealer*

In Bridge, the cards are shuffled by the player to the dealer's left and cut by the player to the dealer's right. The dealer deals out all the cards one at a time so that each player has 13 cards. In our system, the dealer is responsible for shuffling and dealing the cards to each player. It also saves the four hand's cards (associated with characters) into a buffer for later used to construct the user interface. Since the class Dealer controls the initialisation of the Bridge game, it needs to collaborate with class Player to complete its tasks. Figure 6 shows the Dealer class card.

Class *BidBoard*

A bid specifies a number of tricks and a trump suit or no trumps. The possible number of tricks is from a minimum of 1 (i.e. 7 made altogether) to a maximum of 7 (i.e. 13 made altogether) and the possible trump suits rank as follows: No Trumps (highest), Spades, Hearts, Diamonds, Clubs (lowest). It is also possible, during the bidding, to *pass* a bid by any player, to *double* a bid by the other side or *redouble* the opponent's double. The BidBoard class is used to hold all combination cases in a board. The BidBoard class card are presented in Figure 7.

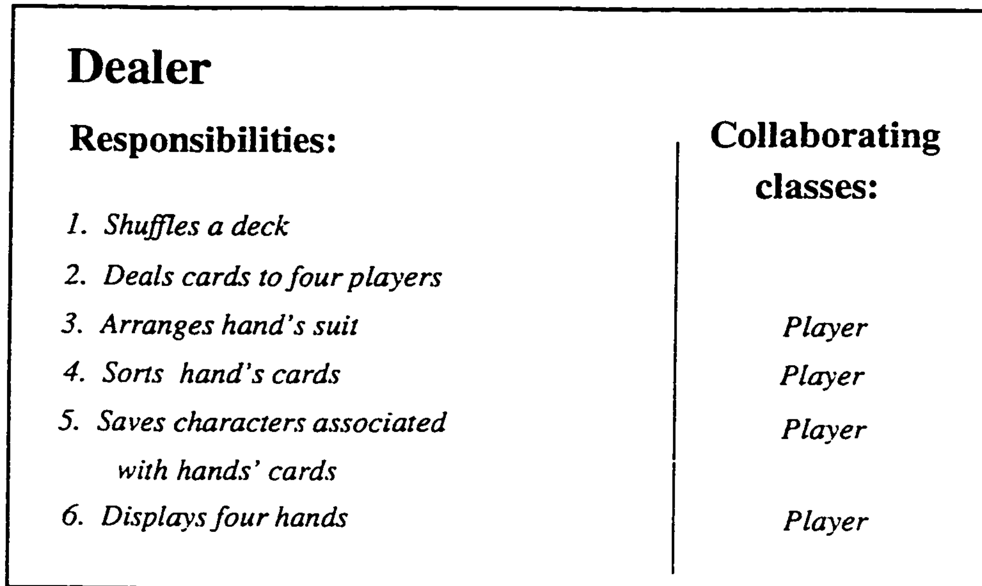

Figure 6: Dealer class card

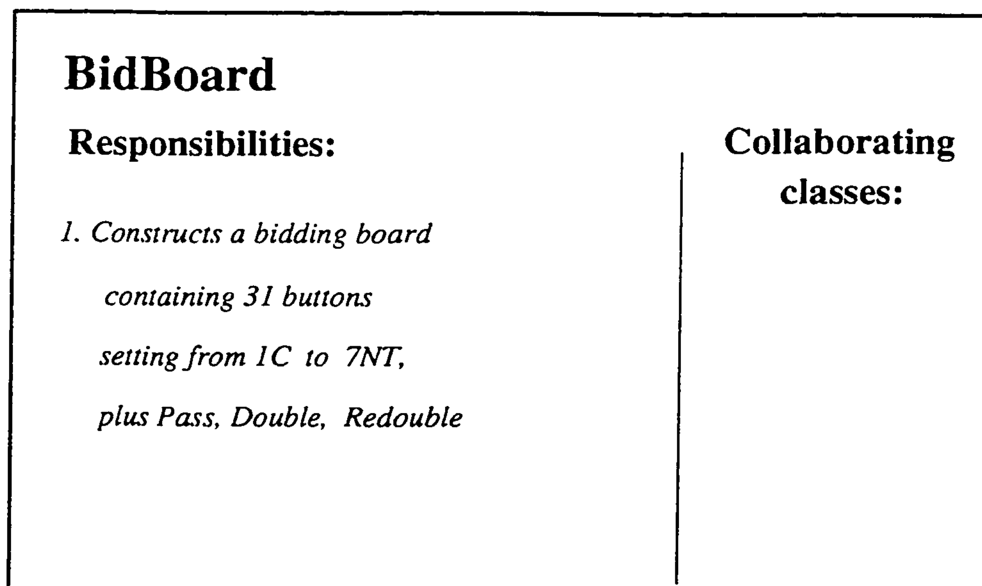

Figure 7: BidBoard class card

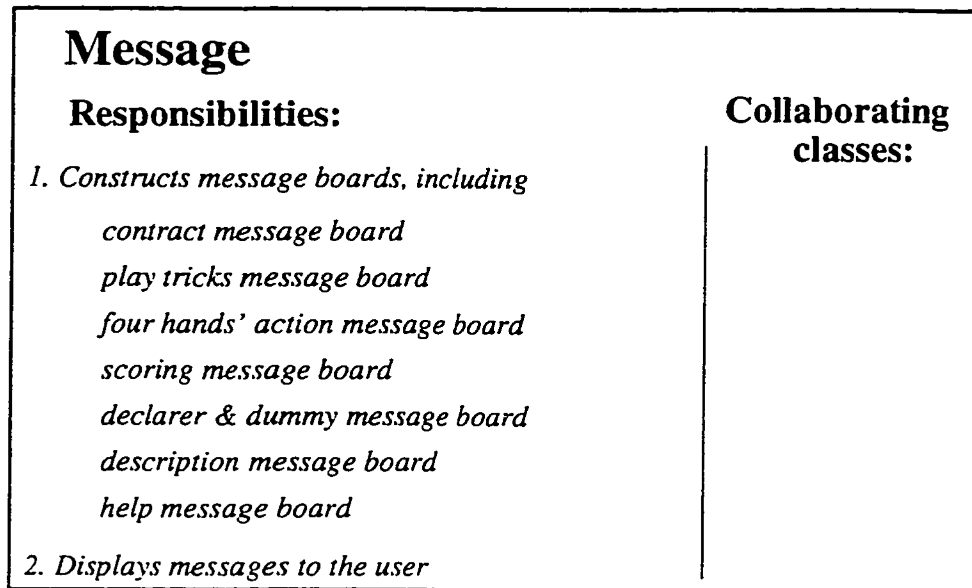

Figure 8: Message class card

Class *Message*
Bridge is a team game. Partners in each side need to communicate with each other through the bidding and play phases to exchange information. What a player bids and what card a player leads have to be announced to the other three players. The scoring is also recorded for each game and displayed in a scoring board. Therefore, we need to create a Message class. The Message class provides the interface for the communication. It simply accepts messages from any class and report these messages to the user. The Message class card is shown in Figure 8.

Class *Command*
A command board is used to issue the commands by a user to the window system (Bridge simulator). While the simulator accepts a command, it will interact with the user to do a certain task. Therefore it is necessary to declare a Command class to represent the command board.

The Command class contains bidding and play commands which are required for the two stages of Bridge. A *new game* command is needed to issue a command to create a new deck once the previous deck has run out. Since the Command class

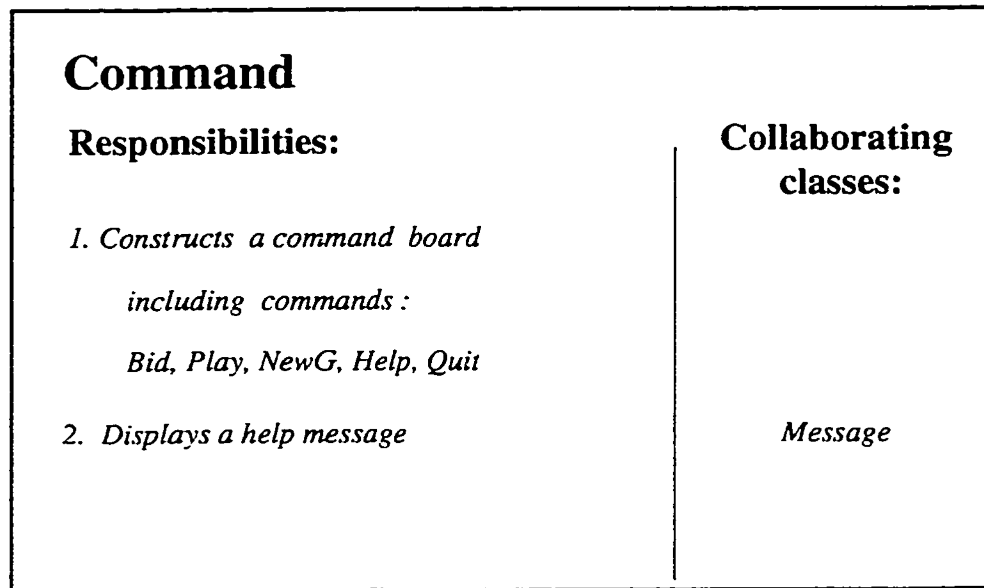

Figure 9: Command class card

can accept a help command from a user and send help messages to the user. it needs to collaborate with the Message class. The Command class is also responsible for accepting a quit command from the user and exiting from the simulator. The Command class card is shown in Figure 9.

Class *HandBoard*

Each hand contains 13 random cards. These 13 cards are constructed in a board which represents an object of the player. Once a player holds 13 cards. the player is available to play a card at each round after the bidding. We call this object HandBoard. and create a class card for it. HandBoard is a class which presents a main part of the interface. and each card in the HandBoard class is a visible portion of the game.

The HandBoard object is completely under the control of the Bridge class (specified later). The HandBoard class has a set of responsibilities. It creates four hand boards in collaboration with the classes Player and Dealer and handles the whole card play process. It is also responsible for initiating various messages to the user in collaborating with the class Message. Figure 10 shows the HandBoard class card.

HandBoard

Responsibilities:	Collaborating classes:
1. Initiates data for a new games	
2. Constructs four hands board	*Dealer*
3. Activates & Deactivates cards	*Dealer*
4. Accepts user's actions	*Message*
5. Handles the card play	*Card, Player, Dealer*
6. Displays each round	*Message*
7. Determines the results	
8. Reposts the scoring	*Message*
9. Destroys old four hands	*Dealer*

Figure 10: HandBoard class card

Bridge

Responsibilities:	Collaborating classes:
1. Initials data	
2. Activates & Deactivates BidBoard	*BidBoard*
3. Accepts user's actions	*Message*
4. Handles the bidding	*Command, HandBoard*
5. Displays a contract	*Message*
6. Sends final contract	*HandBoard*
7. Controls play tricks	*Command, HandBoard*
8. Resets a new game	*Command, HandBoard*
9. Cleans up and exits game	*Command,*

Figure 11: Bridge class card

Class *Bridge*
The Bridge class is a manager that controls the actions of the game as well as being an active participant in the game. It is responsible for bidding and sends a final contract to the HandBoard class. It also keeps track of the play, and the processing of a new game. It collaborates with the classes BidBoard, HandBoard, Command, and Message. The Bridge class card is shown in Figure 11.

4.2.2 Collaboration Graph

As classes begin to be identified and class cards are being developed, it is often useful to visualise how various objects and classes in a system are related. A collaboration graph shows the connections between various classes based on numbered responsibilities supported by various class cards. Figure 12 shows a collaboration graph for the Bridge system based on the notation described in section 3.2.2.

The system contains nine classes which are represented by the boxes. The number in the semicircle is marked with the particular responsibility of the class. The arrow between two classes points to a collaborator which shares a responsibility with the other class to complete a task. A user issues a set of commands to the simulator and controls the system to play the game intuitively.

4.3 Design Bridge User Interface

The design process described in this chapter has considered only the internal structure of the Bridge program. However, the user interface design is also an important part of developing an interactive program. From the beginning, we assumed that the Bridge simulator would have a mouse-driven interface and that the user would issue commands.

4.3.1 Basic Considerations

Window Layout
The consideration of our user interface design is based on the situation of a real Bridge game. Four players sit around a table, West and East face each other and so

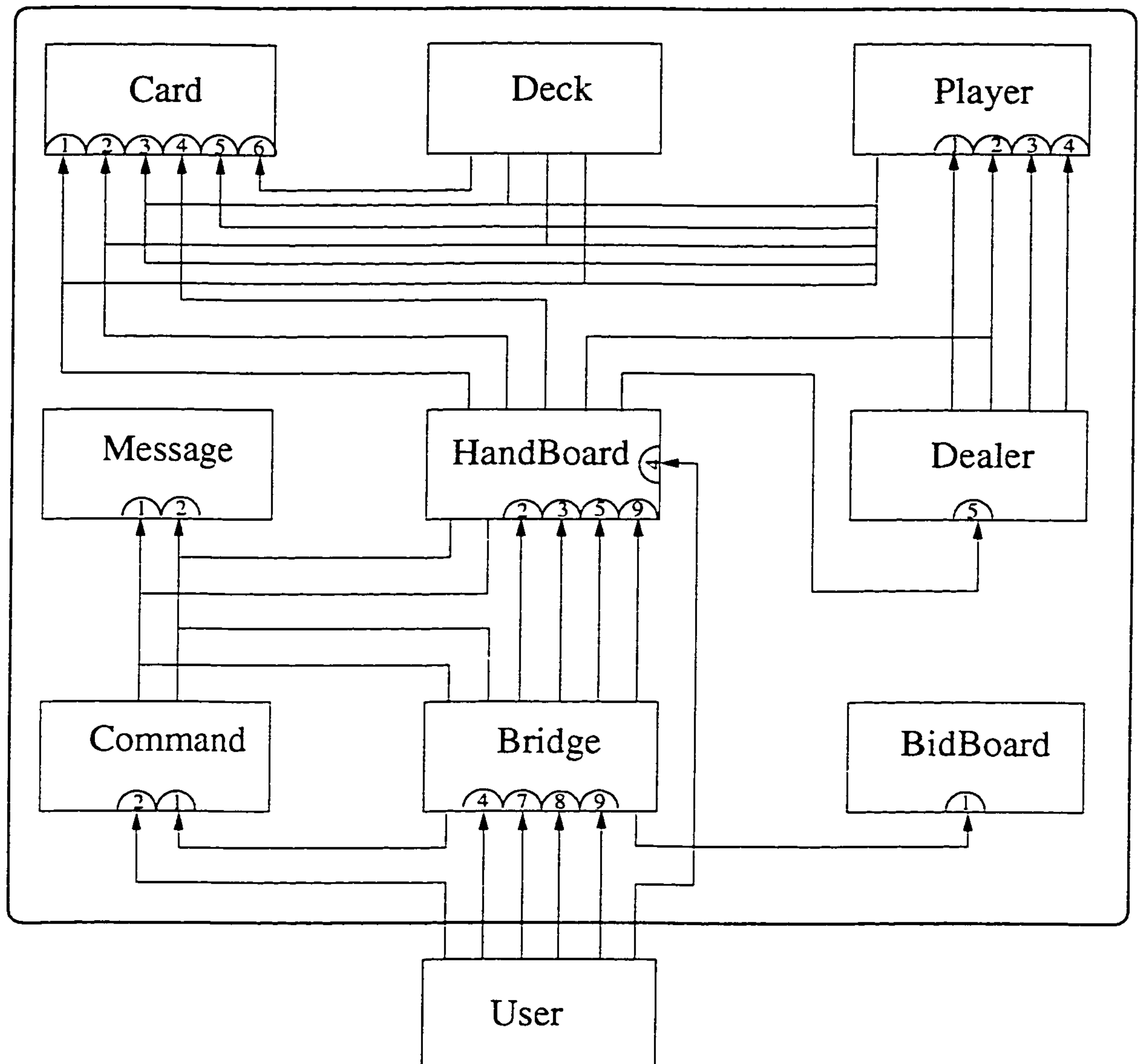

Figure 12: The Bridge collaboration graph

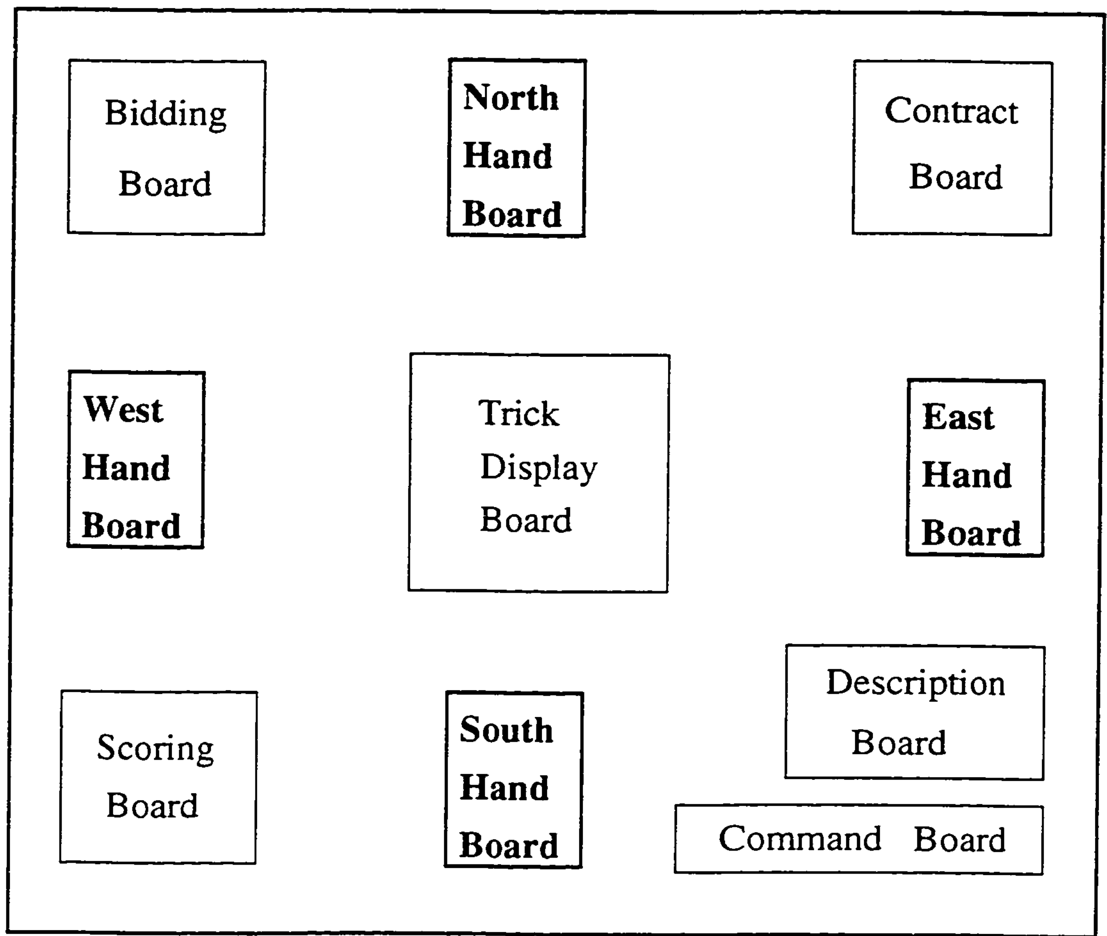

Figure 13: Bridge simulator user interface layout

do North and South. Similarly, we design a Bridge simulator to model four players. West player is on the left middle side of the window. East player is on the right middle side. North player is on the top middle side, and South player is on the bottom middle side. We call the four areas a *hand board.* There is a bidding board placed on the left top side of the interface which contains all the bidding possibilities. The result of the bidding is called a contract and is shown on the right top side of the interface. The center part of the interface displays the 13 tricks of the play. The left side of the bottom gives the scoring of each game and the right side of the bottom is a command board and a set of description for the game. The command board contains commands which allow a user to start the bidding and play; to generate a new game; to get help (not a complete reference manual); and to exit from the program. Figure

13 shows a window layout of the Bridge simulator.

Handling Input

To play Bridge on a computer, the input issues must be handled in the computerised version. The most straight forward interface would allow a user to simply click on a square to make an action. In this case, selecting a square would be similar to the action of pressing a button. Motif allows user to abort a command by moving the cursor outside an armed button before releasing the mouse button. We represent 52 cards in the four hands' boards, 31 bidding possibilities in the bidding board, and 5 commands in the command board as Motif buttons and implemented as Motif push button widgets. A user only needs to push the leftmost button of the mouse to issue a command or to active a card or a bidding board possibility.

Providing User Feedback

It is important for any interactive program to provide effective feedback to a user about the program's current state, and the effect of the user action. If a programmer ignores the possibility of incorrect input, the application may become confused and crash. An approach, in our design, is to disable input in buttons that have already been marked. If input is disallowed, the game should provide some indication that the buttons cannot be chosen. This prevents a player from erasing or writing over the top of an already marked button.

In the system, we plan to design 9 message boxes used to store information related to the different stages of the game and used to report the messages related to a user's action. Here are the description for each of them. Three message boxes are used to store a final contract, a process of play, and a final score of the game individually. On the top of the each player's board, there exists an area used to announce a message to the user or to issue a warning message related to a wrong action. Two other message boxes will indicated a declarer and a dummy players at the bottoms of two player's boards after the bidding.

Chapter 5

Implementation

This chapter presents an implementation of a Bridge simulator based on the design developed in the previous chapter. The task consists of implementing C++ classes that meet the requirements discussed in chapter 4 and determining how these classes fit together to create a working program. Before discussing the implementation of each class in Bridge simulator, we should look at some issues related to the overall structure of the simulator and strategies for programming.

5.1 Implementation Issues

The programming language used for the implementation is C++ with the Motif toolkits for X window management, and the system is implemented under UNIX environment.

5.1.1 Implementation Languages

There are two principal issues that must be resolved if we are to use Motif effectively with C++. The first is the question of how to call functions in the C libraries, such as Motif, from programs written in C++. C++ is an object-oriented version of C. It is compatible with C, so that existing C code can be incorporated into C++ programs. An important feature of the C++ language is that it is simple to call existing C functions from C++. When programming in C++, all the familiar libraries, like Xlib and Motif, are still available.

The second issue is the combination of C++ classes with Motif widgets, particularly when using the object-oriented features of C++. Programming in C++ often works with object-oriented libraries by creating new subclasses of various classes in the library, specialising them to suit the needs of their applications. Programmers who have used this approach may wonder how they can create a C++ class that is a subclass of a Motif widget. The obvious answer is that they cannot. However, Young [14] found a strategy that is to use C++ to create higher level user interface components that combine one or more widgets into a logical grouping. The goal of this approach is not to wrap individual widgets in separate classes. Instead, the intent is to implement the key elements of an application and its interface as C++ classes, using Motif widgets as primitives. We can call Xt and Motif functions to create primitive Motif widgets.

This approach addresses object-oriented programming with C++ and assumes that the classes in a program represent the architectural elements of the application. Classes that have a user interface component use Motif widgets as primitive elements from which to construct that interface. Not everything in a C++ program has to be a class, and it is possible, and sometimes useful, to take a functional approach in portions of a program even when using C++.

5.1.2 User Interface Components

We use the techniques described above for creating user interface components by encapsulating a collection of widgets within an object. The mechanics of creating widgets, specifying widget locations and other resources, assigning callbacks, and so on can all be captured in C++ classes.

We refer to such classes as user interface *components*. A component not only encapsulates a collection of widgets but defines the behaviour of the overall component as well. Widget sets like Motif provide simple, low level building blocks, like buttons, labels, and text fields. Components are C++ classes that use these simple building blocks to create higher level objects like command panels, menus, and so on. In our program, we define such kinds of components as Command class, BidBoard class, and so on.

5.1.3 Problems in C++ with Motif

The question is how one can use member functions of classes as callback functions. A callback function is a function that is called by a widget when it knows you are interested in some events that happened in the outside world. Callback pose a minor problem for C++ classes. C++ member functions have a hidden argument, which is used to pass the *this* pointer to the member function. This hidden argument, which is typically passed as the first parameter, makes ordinary member functions unusable as callbacks for Xt-based widgets. If a member function were to be called from C (as a callback), the *this* pointer would not be supplied, and the remaining arguments would be incorrect. If we want to use member functions to define the behaviour of a C++/Motif user interface component, we have to arrange another way for a member function to be called as a result of a widget callback.

There are several approaches to solving this problem. Our solution is to use *static member functions*. A static member function is similar to a friend function in that it is a regular function that does not expect a *this* pointer when it is called. However, a static member function is a member of a class. It has the same access privileges as any other member function. It can also be encapsulated so it is not visible outside the class. Obviously, some additional code is needed to make this do something useful. Code has to be added to initialise the toolkits, etc.

5.1.4 Programming Process

In C++, all functions must be declared before they are referenced in any way. In the Bridge simulator program, each class is built in three steps. We first declare a class that we are going to use in the program. The declaration phase involves a careful description of the classes' member data and the functions that will accomplish what it is supposed to do. The next step is to define the class, describing in detail how the member functions work and how they are used to communicate between the objects that will comprise our program. Finally, we write the program that will use the classes we have declared and defined.

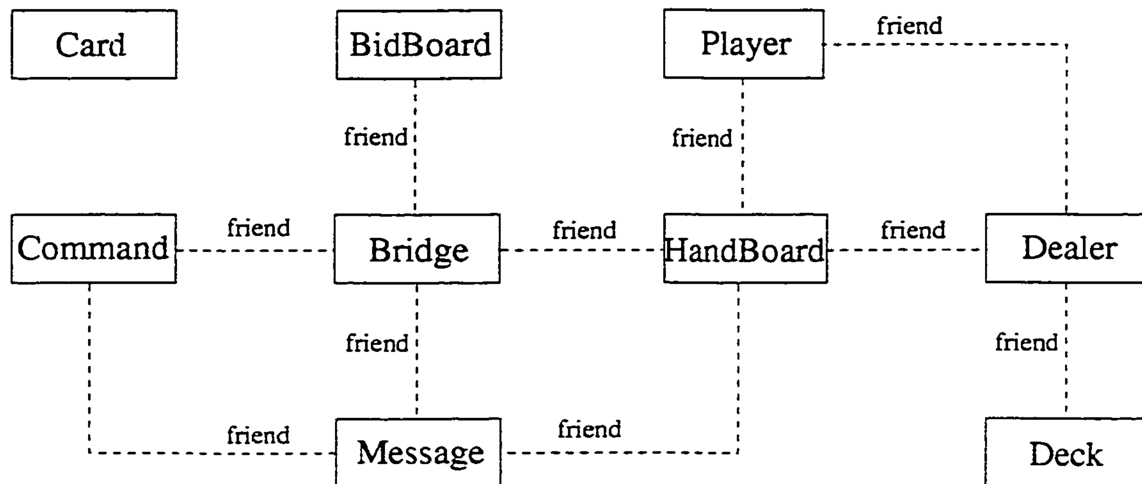

Figure 14: Class structure diagram

5.2 System Structures

Classes structure

The class structure shows the relationships between the classes. The only technique which is used in the Bridge simulator program is the instances of the classes rather than its inheritance because it is not necessary for the application. However, some classes have to cooperate with the member data and private functions of other classes to complete their tasks, so we need to define these classes as *friend* classes of the collaborated classes. Figure 14 illustrates the class structure of the simulator system. Each box represents a class. A class with a dash line represents a friend class of the other one.

Motif Widget Structure

A Bridge simulator is a user interface which consists of a set of basic widgets (such as buttons, labels, etc.) managed by a set of container widgets (such as form, frame, etc.). From the Motif widgets diagram, we can understand how the user interface is constructed and how an object encapsulates a set of widgets. We have pointed out

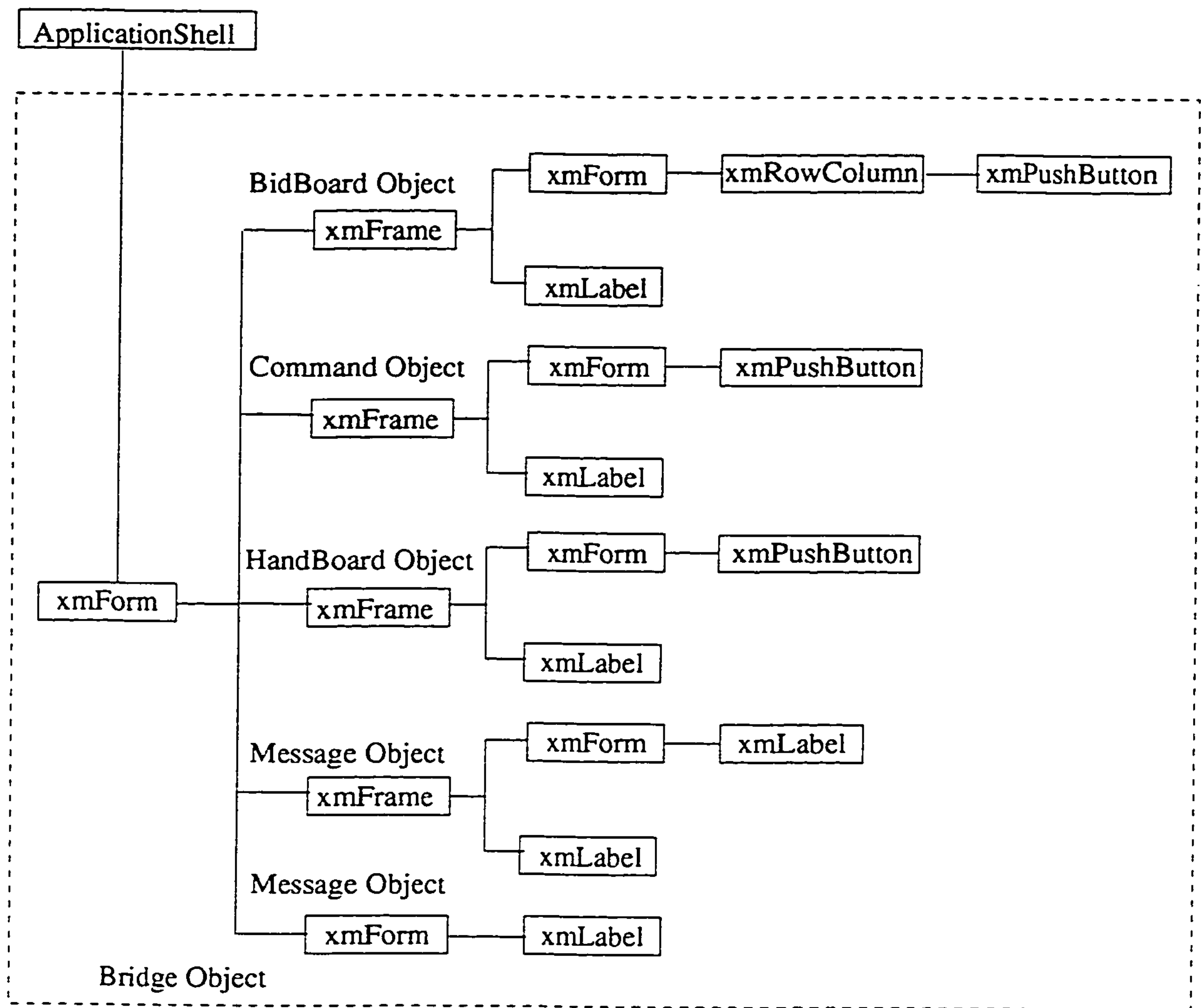

Figure 15: Motif widget structure diagram

the techniques for creating user interface components by encapsulating a collection of widgets within an object. In figure 15. we see some objects encapsulating the same collection of widgets.

System Callback Structure

Bridge class is the key class of the system. It consists of other objects of the classes to build up a user interface, and it controls the command boards to play the game by blocking and activating the command buttons. Figure 15 shows the details of the game playing process by using callback functions so that we can realize how the

Bridge simulator works.

System callback structure is described as the following.

- Initially, except *Bid*, *Help*, and *Quit* buttons on the command board, other buttons on the window are all blocked.
- Once clicking *Bid* command button, the buttons on the Bidding board are activated. After bidding, these buttons are blocked.
- Once clicking *Play* command button, all buttons on the four hands' board will be activated in the turn.
- Once clicking *NewG* (new game) command button, the Play command button is blocked and the Bid command button is activated.

5.3 Features of Windows

Bridge simulator consists of a main window and a dialog pop-up window. The dialog pop-up window contains short messages for guiding using the system. Once a user clicks the *Help* button, a small window will be popped up on the main window at the bottom right corner. The main window has the following features.

- On the top of each hand, there is a message area to display the hand's bidding and card play. It also reports the warning or error messages to indicate a wrong action by the user.
- During the bidding process, after a round, bidding of each hand will be shown on the contract board. The final contract of a deck is displayed on the board until a new game starts.
- When the bidding is finished, the declarer and the dummy hands will be indicated at two player's boards.
- During the card play, after a round, the trick is displayed on the trick display board until 13 tricks are taken out. The bottom of the board also indicates whether the contract is made or whether there are undertricks or overtricks and how many.

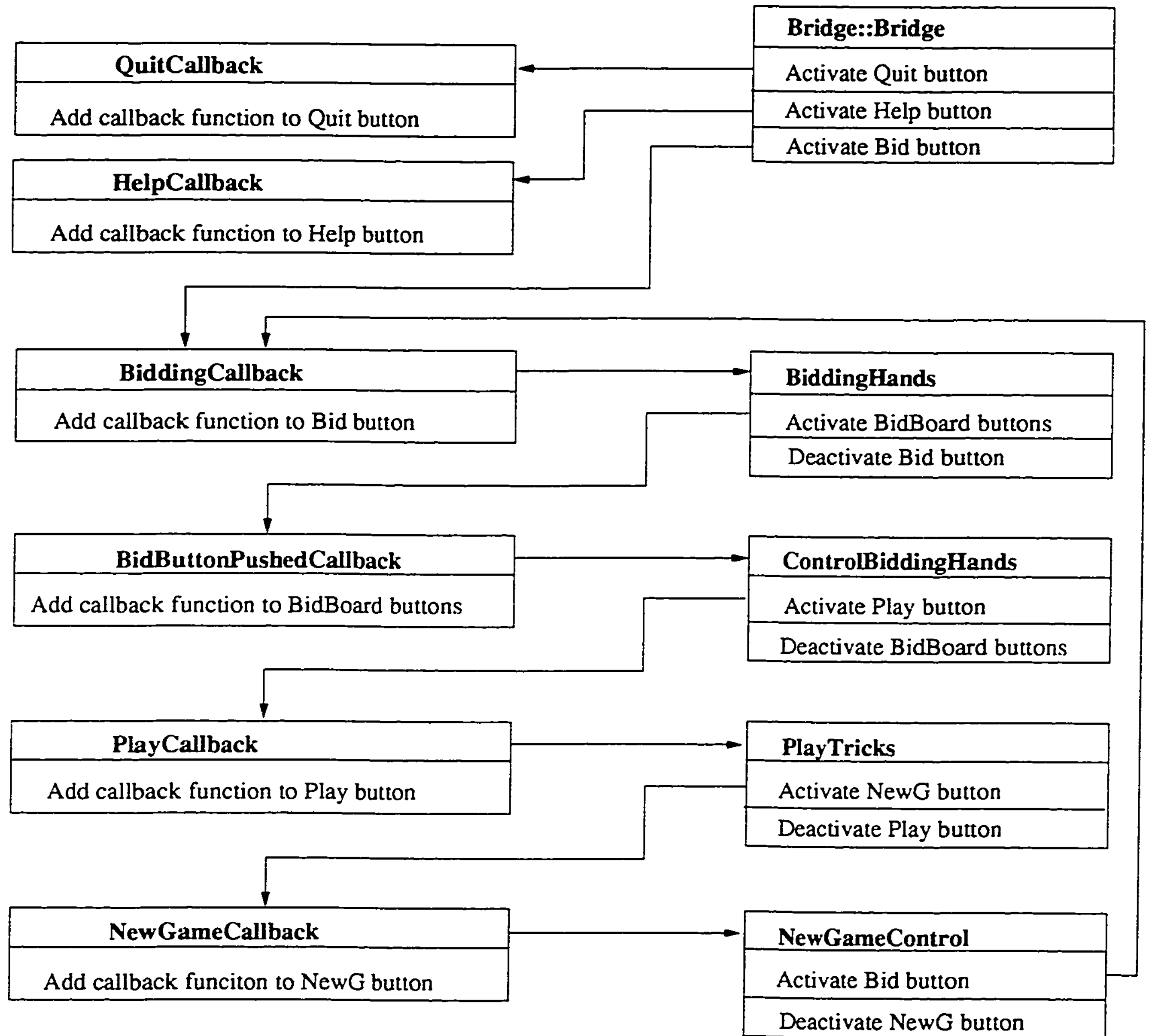

Figure 16: System callback structure

- After the card play, the scoring is reported in detail (such as, who is declarer? What is the contract? Contract is made or not, and what is the score? ect.) on the scoring board. All decks' scores (at most eight decks) are listed on the board until the window is closed.

- A hand's buttons will be activated once the hand turns to lead a card. All buttons in the hand will be blocked after a card has been taken out. The button label will be marked by a character "—" when a card is taken out by a hand.

- Window can be resized by placing the mouse pointer on the small square on the right top of the window, pressing the left button of the mouse, and dragging the mouse to the desired size then releasing the mouse button.

5.4 Overview of Code

Each class in the Bridge simulator has a set of functions (public and private) to carry out certain tasks. We summarise the main member functions along with a brief description in associated classes.

class Card

- *SetVal/GetVal:* Sets/Returns a numeric value (2-14) of a card.
- *SetSuit/GetSuit:* Sets/Returns a suit (S/H/D/C) of a card.
- *SetHCP/GetHCP:* Sets/Returns a High Card Points (HCP) of a card.
- *SetMark/GetMark:* Sets/Returns a mark (1/0) of a card.
- *CardToChar:* Defines a character of a card value.

class Deck

- *Deck:* Initialises a new deck of 52 cards including assign a numeric value (2-14), a suit (S/H/D/C), and a HCP value (0-4).

class Player

- *Player:* Initialises private data of the class object.
- *HandValue:* Determines the High Card Points (HCP) of a player's cards.
- *HandSuits:* Rearranges the suit of a hand as the rank Spade, Heart, Diamond, and Club.
- *SortHandSuits:* Sorts numeric value of each suit in an descending order.
- *HandCardToChar:* Saves character string associated with a hand's cards value into a butter.

class Dealer

- *Dealer:* Generates an object of the Dealer class. It contains the following member functions.
- *Shuffle:* Shuffles the deck by switching each card with a randomly selected card.
- *DealHand:* Deals a deck to the four hands: Calls the member functions of the class Player.
- *PublicHandChar:* Saves character string associated with four hand's cards value into a buffer. it is convenient for constructing button widgets.
- *DisplayHand:* Displays four hand's cards on the screen.

class BidBoard

- *BidBoard:* Generates a bidding board containing 31 buttons to represent all cases of bidding.

class Command

- *Command:* Generates a command board which contains commands *Bid*, *Play*, *NewG*, *Help*, and *Help*.
- *CreateHelpDialogWindow:* Generates a pop-up window when a user click *Help* button. The window contains information for using the simulator.

class Message

- *Message:* Generates message areas, including four hand's message areas, a contract display board, a trick display board, a scoring display board, a dummy and a declarer hand message area, a help message display board, and a description message board.

- *PostMessage:* Converts the character string to a compound string for Motif and displays the compound string on the label.

class HandBoard

- *HandBoard:* Generates 13 buttons (represent 13 cards) for each hand's board and set values on the buttons.

- *~HandBoard:* Destroys the objects of collaborating classes.

- *InitialData:* Initializes private data

- *DestroyFourHands:* Destroys previous four hand's button widgets when a new game starts.

- *GetFinalContract:* Finds a final declarer, a doubler, a redoubler, and a contract.

- *ActivateHands/DeactivateHands:* Adds/Removes callback functions to the buttons of hands, make buttons activated/deactivated.

- *ControlHands:* Controls game play process by calling several functions.

- *CheckCardsInHand:* Checks a "legal" card leaded in a player's hand.

- *CheckCardsInSameSuit:* Checks a card leaded in the same suit as the first leader's of the round unless no more left in the hand.

- *NextLeadHand:* Finds next round leading player by examining current round cards.

- *OutputDisplay:* Displays each round in the trick display board.

- *WinTricks:* Records tricks won by the declarer.

- *PrintScoring:* Calculates score and display it in the scoring board.

class Bridge

- *Bridge:* Generates a Bridge object, including a bidding board, four hands board, a command board, a contract board, and a description board; Activates the *Quit* and the *Bid* command buttons.

- *~Bridge:* Destroys objects which collaborate with Bridge object.

- *InitialData:* Initialises the private data of the Bridge object.

- *BiddingHands:* Finds the first bider and posts it; Calls function *ActivateBidBoard* to make first bider's hand buttons activated; Deactivates the *Bid* command button.

- *ActivateBidBoard/DeActivateBidBoard:* Adds/Removes callback functions to the 31 buttons in the bidding board, make buttons activated or deactivated.

- *ControlBidHands:* Controls the process of bidding by calling several functions: If no contract for bidding, activates the *Bid* command button, otherwise activates the *Play* command button; Finds a first leading player by calling function *FindContract*.

- *FindContract:* Determines a final contract.

- *DealBidding:* Checks input (31 bidding board buttons) whether is an "legal" or "illegal" bid, then saves a bidding if it is "legal".

- *BidOutputDisplay:* Displays bidding of each round in the contract board.

- *PlayTricks:* Sends a final contract to the object of the HandBoard class; Deactivates the *Play* command button and activates the *NewG* command button; Posts the first leading player and activates the leading player's hand board buttons.

- *NewGameControl:* Calls the function *DestroyFourHands* of the HandBoard class to destroy previous four hands and generates new four hands; Activates *Bid* command button and deactivates the *NewG* command button; Clears four hand's messages and resets the size of contract board and trick display board.

- *QuitWindow:* Closes window and exits from the program.

Chapter 6

Results

In this chapter. we describe the results that the Bridge simulator is capable of producing. We summarise the performance of the simulator first. then present and describe examples of bidding and play.

Simulator Performance

The result of this project was a user interface which simulates the whole process of Bridge card play. The user interface is an interactive application which can be used to teach people to play Bridge or it can be used as a simulator for people to study and analyse the game of Bridge so that they can get more experience for bidding and play. The performance of Bridge simulator include the following aspects:

- The action of the simulator might bring people to a better intuitive understanding of the game of Bridge through the main simulator window.
- The simulator is a mouse-driven interactive user interface. which people find easy to use.
- The simulator can generate different decks. For each deck the cards are randomly distributed to each hand.
- The simulator manages the processes of bidding and play of the Bridge and handles various messages at different stages.
- Only one of *Bid*. *Play*. and *NewG* (new game) buttons can be activated at a time. Buttons on the Bidding board can only be activated at bidding stage and buttons on the four hands' board can only be activated at play stage.

Table 1: Abbreviation for bidding board

Abbreviation	Bidding
1C/2C ... 7C	One Club/Two Clubs ... Seven Clubs
1D/2D ... 7D	One Diamond/Two Diamonds ... Seven Diamonds
1H/2H ... 7H	One Heart/Two Hearts ... Seven Hearts
1S/2S ... 7S	One Spade/Two Spades ... Seven Spades
PS	Pass
DB	Double
RD	Redouble

Table 2: Abbreviation for hands' board

Abbreviation	Cards
2C/3C ... TC/JC/QC/KC/AC	2♣/3♣ ... 10♣/ J♣/Q♣/K♣/A♣
2D/3D ... TD/JD/QD/KD/AD	2♢/3♢ ... 10♢/ J♢/Q♢/K♢/A♢
2H/3H ... TH/JH/QH/KH/AH	2♡/3♡ ... 10♡/ J♡/Q♡/K♡/A♡
2S/3S ... TS/JS/QS/KS/AS	2♠/3♠ ... 10♠/ J♠/Q♠/K♠/A♠

Examples of bidding and play of the simulator

Before we present how the Bridge simulator works by two typical examples, we will show the notions which we used on the window by the two tables: Table 1 and Table 2.

Example 1

The first example will present a bidding process of the Bridge game. Figure 17 shows a situation with four hands' cards of a deck displayed on the hands' area and a final bidding contract displayed on the contract area.

The goals of bidding are: How much (to what level) to bid and where to bid (which suit, no trump or a trump suit) to choose. In the example, the procedures for bidding are:

- Dealer (North) calls first; Bidding proceeds clockwise.

- To bid, a player must make a higher bid than the preceding one (A bid is higher if it contracts for more tricks, or for the same number of tricks in a higher ranking suit.). In figure 17, such as North called two Diamonds after West called two Clubs and after East called two No Trump, West called three Spades.

- The bidding ends after three consecutive passes, such as South, West, and North all bid Pass after East bids four Spades (Exception: if everyone bids Pass in the first round, then the game ends.).

In the example, the last bid (4S) of the bidding becomes the final contract. East and West have make a commitment to take ten (4+6) tricks with Spade as trump suit. At this point, one player of the side bidding the final contract takes over and enjoys the responsibility of playing the cards and tries to win tricks: that player is called the declarer. Declarer's partner, who has nothing further to do, is called the dummy. The declarer is the one for the side winning the final contract who first named the suit (trump suit or no trump) of the final bid. In this example, East is the declarer, because East had the first bid on one spade for East-West. West is the dummy. North and South players are called defenders.

Example 2

The second example will present the process of play in the Bridge game. In figure 18, the trick display area (the center of the window) shows thirteen tricks were played by the hands. The final score was also displayed on the scoring board.

The goals of play are: Declarer tries to fulfill the contract made in the bidding and the defenders try to prevent declarer from fulfilling the contract. In the example, the procedures for play are:

- Declarer (East) is the member of the partnership bidding the final contract. Dummy (West) is declarer's partner.

- South, the opening leader (player who leads to the first trick), is the opponent to declarer's left.

- Opening leader plays any card. Dummy exposes its entire hand. Play proceeds clockwise: the player who wins a trick leads to the next.

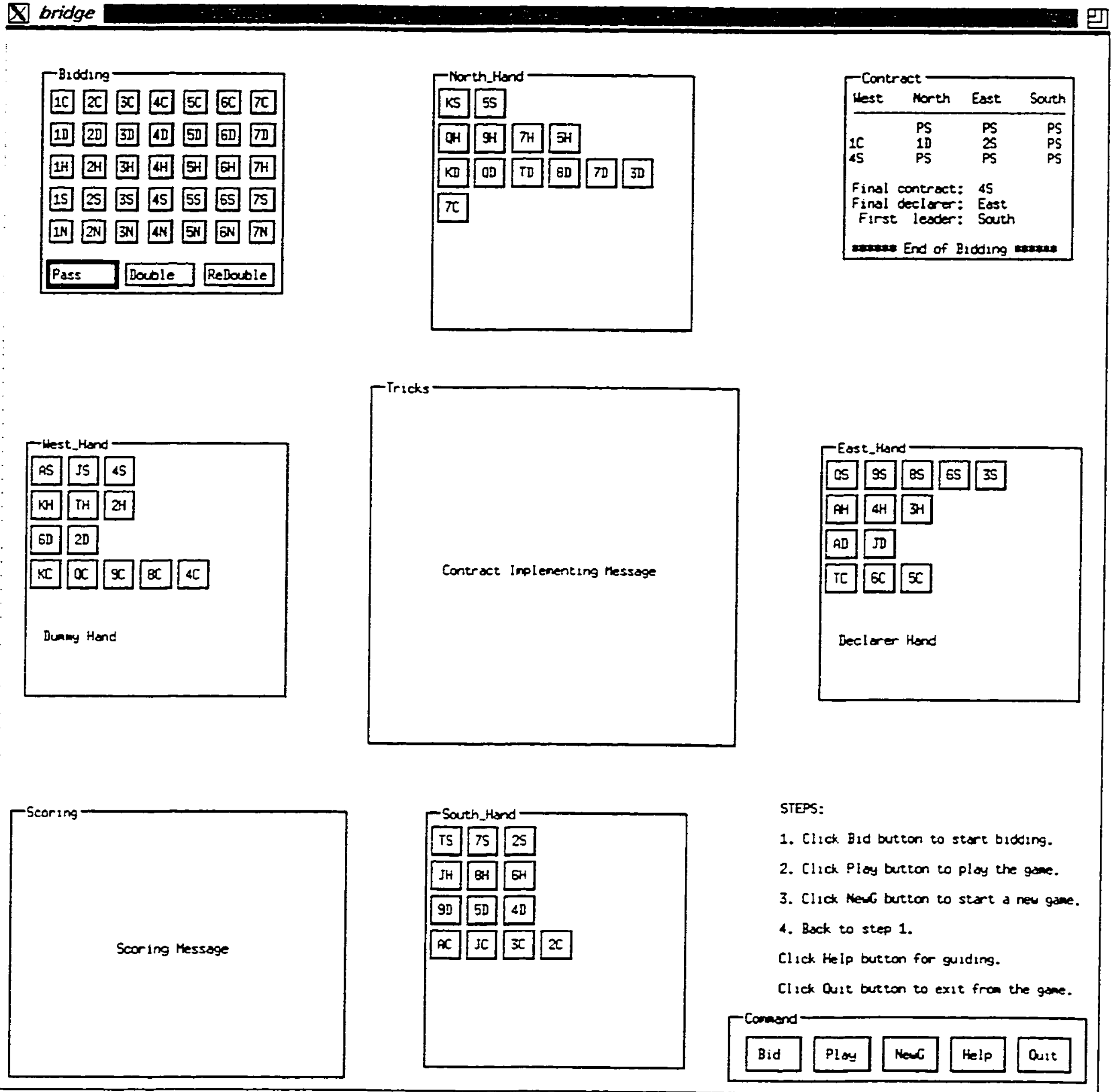

Figure 17: Example 1

Play at Bridge consists of tricks. Each player in turn, proceeding clockwise around them, removes one card from his or her hand and places it on the top of the hand's message area. When all four players have played, the four cards displayed in the trick area constitute a trick. Since each player has thirteen cards, and since a trick consists of one card contributed by each player, there are a total of thirteen tricks in each deal.

bridge

Bidding

1C 2C 3C 4C 5C 6C 7C
1D 2D 3D 4D 5D 6D 7D
1H 2H 3H 4H 5H 6H 7H
1S 2S 3S 4S 5S 6S 7S
1N 2N 3N 4N 5N 6N 7N
Pass Double ReDouble

North_Hand

Contract

West	North	East	South
	PS	PS	PS
1C	1D	2S	PS
4S	PS	PS	PS

Final contract: 4S
Final declarer: East
First leader: South

****** End of Bidding ******

West_Hand

Tricks

Tricks:	(Leader)	West	North	East	South
01	South	6D	KD	AD	4D
02	East	AS	5S	3S	2S
03	West	4S	KS	6S	7S
04	North	2D	QD	JD	5D
05	North	4C	7C	5C	AC
06	South	TH	QH	AH	6H
07	East	JS	3D	8S	TS
08	West	KC	7D	6C	2C
09	West	QC	5H	TC	3C
10	West	8C	8D	9S	JC
11	East	KH	7H	3H	8H
12	West	9C	TD	4H	JH
13	West	2H	9H	QS	9D

West and East win: 10
W & E make: 10

****** End of Play ******

East_Hand

Scoring

Board	Declarer	Contract	DB/RDB	Result	Score
1-NO	East	4S	N/N	Make	120

South_Hand

STEPS:

1. Click Bid button to start bidding.
2. Click Play button to play the game.
3. Click NewG button to start a new game.
4. Back to step 1.

Click Help button for guiding.

Click Quit button to exit from the game.

Command

Bid Play NewG Help Quit

Figure 18: Example 2

Chapter 7

Conclusion

In this chapter, we conclude the Bridge simulator system and summarise the advantages of using object-oriented methodology and language (C++) for design and implementation of Bridge simulator application. We also discuss a research on computer Bridge and suggest avenues for the further directions in which the project could be extended.

7.1 Bridge Simulator System

In this report, we present a Bridge simulator with object-oriented design and programming. As design is the most important phase in object-oriented software development, Bridge simulator aims at providing automated support to illustrate how we apply the concepts of object-oriented to solve a real problem. Important features of the Bridge simulator are using the object-oriented paradigm to design and using X Window/Motif toolkits to construct an interface for simulating the bidding and playing of Bridge.

7.1.1 Experience on Object-Oriented Programming

Object-oriented programming offers a new and powerful model for writing computer software. Object-oriented approach speeds the development of new programs, improves the maintenance, reusability, and modifiability of software. We will summarise how object-oriented method works in the Bridge simulator software design and programming.

- Object-oriented approach provides a powerful methodology by using the same language to deal with analysis, design, and programming within the entire Bridge simulator software development process. It reduces the complexity of the system architecture design and implementation.

- Object-oriented design supports abstraction at the object-level. Abstraction allows us to ignore the details of a problem and concentrate on the whole system. In the Bridge simulator, we found out the objects of Bridge domain and defined those objects by the classes such as Card, Deck, Player, and Dealer etc.

- Each class in the Bridge simulator stands by itself within a module. The functions and attributes within the class are held together cohesively by the object which they are modelling. Therefore, some classes, such as Card, Deck, Player, and Dealer, can be used by any card game without change or a little change for their purposes.

- Object-oriented methods make data maintenance easier to manage because data is hidden behind the class interface in the object-oriented design. Changes in the data effect only one class at a time and can carefully be managed and tested.

7.1.2 The Advantages of C++

C++ is an object-oriented language. We have used it to implement our Bridge simulator. Below main features of C++ are given [2].

- C++ implements *data abstraction* in a clean way using a concept called *classes*. Data abstraction is a way of combining data with the functions used to manipulate the data so that implementation details are hidden from the programmer. Data abstraction makes programs much easier to maintain and upgrade.

- C++ makes parts of programs easily reusable and extensible. This is where the word *object* comes from. Programs are broken down into reusable objects. These objects can then be grouped together in different ways to form new programs. Existing objects can also be extended.

- C++ makes existing programs easily modifiable without actually changing the code. This is a unique and very powerful concept. The two new concepts,

inheritance and *polymorphism*, have been applied. The existing object stays the same, and any changes are layered on top of it. The programmer's ability to maintain and adjust code without introducing bugs is drastically improved using these concepts.

- C++ has a feature called *operator overloading*. This feature lets a programmer specify new ways of using standard operators in his/her own programs. The overloading concept extends to all functions created in C++.

- C++ also cleans up the implementation of several portions of the C language, most importantly I/O and memory allocation. The new implementations have been created with operator overloading, so that it is easy to add new types and provide I/O operations and memory allocation for them.

7.2 Computer Bridge

Computer Bridge has attracted a considerable amount of attention from computer games researchers. However, researchers have not succeeded to produce a system whose performance is capable of competing with people because Bridge itself is hard and is quite different from the other games such as chess.

Gambäck and colleagues [6] summarise the basic problems that make Bridge a hard game to play. First, it is necessary at almost every stage to reason about the other players' knowledge and beliefs. Second, most knowledge is probabilistic in nature. The points above apply equally to bidding and play; however, for bidding it is at least possible to achieve a certain level of performance with conventional programming methods. For card play, there is the added problem of having to be able to construct complex plans and counter plans. Human analysis of card play situations almost invariably begins by constructing plans for the play in each suit individually, and then uses these as building blocks to construct a plan for the play of the entire hand. This type of analysis is considerably more difficult to model than the tree searching algorithms used by, for example, most chess programs.

Some people have claimed that computer Bridge is hard. For example, Brent Manley, Bridge expert and reviewer of Bridge programs for the magazine of the American

Contract Bridge League, has said about Bridge [8]: *It is generally accepted that writing a computer Bridge program that plays Bridge well is nearly impossible.*

Tom Throop, one of the leading producers of commercial Bridge software, has also said about the prospects of computer Bridge [12]: *The task of writing a Bridge-playing program that demonstrates a high level of intelligence is actually more difficult than writing a similarly intelligent chess program.*

In general, it is difficult to simply apply existing game playing techniques to the game of Bridge because of the incomplete nature of the information available to the players. Unlike in chess, where the positions of all the pieces are known, in Bridge you can only guess which cards your opponents hold [5]. Frank has looked at the literature on the game of Bridge, examining both academic research and commercial products and concluded that although expert level performance has been achieved in the sub-problem of Bridge bidding, the overall standard of computer Bridge play in general is very low. This contrast sharply to the success of computers in other games [5].

7.3 Further Work

In this report, we present the design and implementation of a Bridge simulator with object-oriented methodology and Motif toolkits. The limitation for the Bridge simulator is only one person can use it at a time because four player's cards are contained by one window. Therefore we suggest the further work could be considered in the following directions:

- Simulating a duplicate Bridge by running two simulators with synchronism. The game is basically the same but the element of chance is reduced by having the same deals replayed by different set of players (Eight players are required.). There are some significant differences in the scoring.

- Constructing a network module with four computers connected together. Each player controls a machine and communicates with others by sending and receiving messages through the network. Therefore four people can play the game at a time by using the different computers.

- Computer Bridge. It requires to write a computer bidding or computer play program that competes with people. The game is played by a player and a computer which represents other three players.

Bibliography

[1] Grady Booch. *Object-Oriented Analysis and Design with applications.* The Benjamin/Cummings Publishing Company. Inc. 1994

[2] Marshall Brain and Kelly Campbell. *Understanding C++: An Accelerated Introduction.* http://www.iftech.com/oltc/cpp/cpp0.stm. 1997

[3] Kent Beck and Ward Cunningham. *A Laboratory For Teaching Object-Oriented Thinking.* OOPSLA '89 Proceedings

[4] Rick Decker and Stuart Hirshfield. *The Object Concept An introduction to Computer Programming Using C++.*

[5] Ian Frank. *Computer Bridge Survey.* Japan 1997

[6] Björn Gambäck. Manny Rayner and Rarney Pell. *Pragmatic Reasoning in Bridge* University of Cambridge. Computer Laboratory. April 1993.

[7] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley. 1992

[8] B. Manley. *Bridge software: Moving forwards or backwards?* The Bulletin. American Contract Bridge League, October 1994.

[9] Terence Reese and David Bird. *Bridge, the Modern Game.* Faber and Faber Ltd. 1983.

[10] Terence Reese and T. Dormer. *The Play of the Cards.* Robert Hale Limited. 1991.

[11] James Rumbaugh, Michael Blana, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Inc., 1991

[12] T.Throop. *Computer Bridge.* Hayden, 1983.

[13] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Inc. 1990

[14] Douglas A. Young. *Object Oriented Programming with C++ and OSF™/Motif.* 2nd Edition, Prentice Hall PTR, 1995.

Appendix A

card.h

```
//card.h: The declaration of class Card
#ifndef CARD_H
#define CARD_H
enum Suit {C, D, H, S};

class Card  {

  public:
     void SetVal(int);        //Set the card's value (2-14)
     void SetSuit(Suit);      //Set the card's suit (S/H/D/C)
     void SetHCP(int);        //Set HCP to a card
     void SetMark(int);       //Set a marked card
     int  GetVal();           //Retrieve the card's value
     Suit GetSuit();          //Retrieve the card's suit
     int  GetHCP();           //Retrieve the card's HCP
     int  GetMark();          //Retrieve the marked card
     void Display();          //Display a card
     char CardToChar();       //Get a character of a card

  private:
     Suit  cardSuit;          //Card suit
     int   cardValue;         //A value(2-14, represents 2...J,Q,K,A)
     int   cardHCP;           //High Card Points
     int   cardMark;          //T:in the hand; F:remove from the hand
};
#endif
```

Appendix B

deck.h

```
//deck.h: The declaration of class Deck
#ifndef DECK_H
#define DECK_H
#include "card.h"

class Deck  {
  friend class Dealer;

  public:
     Deck();                    //Constructor, initialises 52 cards
     void ShowCard();           //Displays a card

  private:
     int   topCard;             //Points to a top card of deck
     Card  cards[52];           //A deck has 52 cards.
};
#endif
```

Appendix C

player.h

```
//player.h: The declaration of class Player
#ifndef  PLAYER_H
#define  PLAYER_H
#include "deck.h"

class Player  {
  friend class Dealer;
  friend class HandBoard;

  public:
     Player();                              //Constructor

  private:
     int   HandValue();                     //Determine HCP of a hand
     void  HandSuits();                     //Rearrange suits of cards
     void  SortHandSuits();                 //Sort a hand's cards
     void  HandCardToChar();                //Get a string of a card
     void  DisplayHandChar();               //Display a hand's char

     const int NumCards=13;                 //13 cards in each hand
     const int suit=4;                      //4 suits at most in each hand
     int   NumC, NumD, NumH, NumS;          //Num. of cards in each suit
     int   numC, numD, numH, numS;          //Num. of marked cards in each suit
     Card  hand[NumCards];                  //Cards of one hand
     Card  handSuit[suit][NumCards];        //Sorted cards of one hand
     char  handChar[suit][NumCards];        //Character string of cards' value
};
#endif
```

Appendix D

dealer.h

```
//dealer.h: The declaration of class Dealer
#ifndef  DEALER_H
#define  DEALER_H
#include "player.h"

class Dealer  {
  friend class HandBoard;

  public:
     Dealer();                          //Constructor

  private:
     void    Shuffle();                 //Shuffle the deck
     void    DealHand();                //Deal cards to four players
     void    PublicHandChar();          //Copy all cards (chars.) into a buffer
     void    ShowHandNS(Player);        //Display N & S hands on the screen
     void    DisplayHand();             //Display four hands on the screen

     Player  peast;                     //East  player(hand)
     Player  pwest;                     //West  player(hand)
     Player  psouth;                    //South player(hand)
     Player  pnorth;                    //North player(hand)
     Deck    deck;                      //One deck
     char    charbuf[4][4][13][3];      //Four hands' string of cards
     int     hdstnum[4][4];             //Num. cards in each suit of four hands
     int     maxnum;                    //Num. cards in the longest suit
};
#endif
```

Appendix E

bidboard.h

```
//bidboard.h: The declaration of class BidBoard
#ifndef  BIDBOARD_H
#define  BIDBOARD_H
#define  EDGEDS3      30
#define  ROWS         5
#define  COLS         7

class BidBoard  {
  friend class Bridge;

  public:
     BidBoard(Widget);              //Constructor

  private:
     Widget  frame;                 //Frame Widget for bidding board
     Widget  label;                 //Label Widget
     Widget  bidform;               //Form  Widget
     Widget  top;                   //Row Column Widget
     Widget  bottom;                //Row Column Widget
     Widget  subtop[35];            //Push Button Widget
     Widget  subbottom[3];          //Push Button Widget
};
#endif
```

Appendix F

command.h

```
//command.h: The declaration of class Command
#ifndef  COMMAND_H
#define  COMMAND_H
#include "message.h"
#define  EDGEDS      10
#define  DISTANCE    5
#define  HELP        9

class Command  {
  friend class Bridge;

  public:
     Command (Widget);                      //Constructor
     void CreateHelpDialogWindow();         //Create a dialog window

  private:
     //Callbacks registered with widget in the Help button
     static void HelpDialogCallback (Widget, XtPointer, XtPointer);

     Widget  frameCmd;              //Frame Widget for command board
     Widget  labelCmd;              //Label Widget for the title
     Widget  command;               //Form  Widget "command" container
     Widget  Bid;                   //Push Button Widget for bidding
     Widget  Play;                  //Push Button Widget for play
     Widget  NewGame;               //Push Button Widget for a new game
     Widget  Help;                  //Push Button Widget for help
     Widget  Quit;                  //Push Button Widget for exit
     Widget  Dialog;                //Create message dialog
     Message *helpmessage;          //Create an object of class Message
};
#endif
```

Appendix G

message.h

```
//message.h: The declaration of class Message
#ifndef MESSAGE_H
#define MESSAGE_H
#define EDGEDS      10
#define TOPDS       300
class Message   {
  friend class Bridge;
  friend class HandBoard;
  friend class Command;
public:
   Message ( const char*, Widget, int);   //Constructor
   void  HandsBoard    (Widget);          //Build Four hands' board
   void  ContractBoard (Widget);          //Build contract board
   void  ScoreBoard    (Widget);          //Build scoring board
   void  TrickBoard    (Widget);          //Build trick display board
   void  DescripBoard  (Widget);          //Build description board
   void  BoardMessage  (const char*, Widget, int); //Display board message
   void  PostMessage   (char*, int);      //Display message
private:
  Widget  message[11];       //Label Widget for message display
  Widget  subform2[4];       //Form  Widget for hands' board
  Widget  frameCon;          //Frame Widget for contract board
  Widget  labelCon;          //Label Widget for the title of the board
  Widget  formCon;           //Form  Widget "contract" container
  Widget  frameTri;          //Frame Widget for trick display board
  Widget  labelTri;          //Label Widget for the title of the board
  Widget  formTri;           //Form  Widget "tricks" container
  Widget  frameScr;          //Frame Widget for scoring board
  Widget  labelScr;          //Label Widget for the title of the board
  Widget  formScr;           //Form  Widget "scores" container
  Widget  formDes;           //Form  Widget for description board  };
#endif
```

Appendix H

handboard.h

```
//handboard.h: The declaration of class Handboard
#ifndef  HANDBOARD_H
#define  HANDBOARD_H
#include "dealer.h"
#include "message.h"
#define  OUTPUT    5                         //Trick Display board ID
#define  DUMMY     7                         //Dummy hand ID
#define  DECLARER  8                         //Declarer ID
#define  SCORE     10                        //Scoring board ID
#define  EDGEDS2   20                        //Define constants
#define  TOPDS     300

class HandBoard  {
  friend class  Bridge;

public :
  HandBoard  (Widget);                       //Constructor
  ~HandBoard ();                             //Destructor
  void  InitialData     ();                  //Initialise data values
  void  CreateFourHands ();                  //Set up four hands' areas
  void  CreateCardBoard ();                  //Generate four hands' cards
  void  ActivateHands   (int);               //Allow input to cards
  void  DeactivateHands ();                  //Shut off input
  void  GetFinalContract(char*, int, int, int); //Receive a final contract
  void  ControlHands (Widget, int, char*); //Control cards play
  int   CheckCardsInHand(char*);             //Verify cards in the hand
  int   CheckCardsInSameSuit(char,char*,int); //Verify cards'  suit
  int   NextLeadHand    (int);               //Get next leading player

private :
  //Callbacks registered with widgets in the grid
  static void ButtonPushed (Widget, XtPointer, XtPointer);
  void   OutputDisplay     ();               //Display tricks
```

```
    void   HandsLeadDisplay  (char*);        //Display leading card
    void   PostNextLeader    (int);          //Post next leading player
    void   SetButtonNewValue (Widget, int);  //Mark a button with "-"
    void   WinTricks         (int);          //Get scores
    void   PrintScoring      (int);          //Display scores
    void   DestroyFourHands  ();             //Destroy four hands' cards

    Widget  topform;            //Form  Widget for the window
    Widget  frame[4];           //Frame Widget for four hands board
    Widget  subform[4];         //Form  Widget for four hands  board
    Widget  label[4];           //Label Widget for four hands' title
    Widget  grid[4][13];        //Push Button Widgets for 52 cards
    Dealer  *theDealer;         //Create an object of class Dealer
                                //Create objects of class Message
    Message *outmessage;        //Display tricks
    Message *handsmessage[4];   //Display four hands' message
    Message *dummymesg;         //Display "dummy" hand
    Message *declarermesg;      //Display "declarer" hand
    Message *scrmessage;        //Display scoring
    Card    dealcards[4];       //Save cards of each trick
    int     gridSize;           //Size of each square
    int     countPlayer;        //Count number of players
    int     times;              //Count number of tricks
    int     inputSuit;          //C:0, D:1, H:2, S:3
    int     revSuit;            //C:3, D:2, H:1, S:0
    int     comeLeader;         //Next player
    int     trumpSuit;          //0:C,1:D,2:H,3:S,and 4; No Trump
    int     gameTimes;          //Count number of games
    char    leaderSuit;         //Leading suit for first round
    char    FinContract[4];     //A final contract
    int     FinDealer;          //A final declarer
    int     LastDber;           //A final doubler
    int     LastReDber;         //A final redoubler
    char    round[4][4];        //Save each trick
    char    mes[500];           //Save 13 tricks
    char    buf[13][60];        //Save tricks for display
    int     lastLeader[13];     //Save the winner of each trick
};
#endif
```

Appendix I

bridge.h

```
//bridge.h: The declaration of class Bridge
#ifndef   BRIDGE_H
#define   BRIDGE_H
#include  "handboard.h"
#include  "command.h"
#include  "bidboard.h"
#define   CONTRACT     4                //Contract board ID
#define   OUTPUT       5                //Trick Display board ID
#define   DESCRIPTION  6                //Description board ID
#define   DUMMY        7                //Dummy hand ID
#define   DECLARER     8                //Declarer ID
#define   DISTANCE     5                //Define constants
#define   ROWS         5
#define   COLS         7
#define   EDGEDS       10
#define   TOPDS        300

class Bridge   {

public:
  Bridge  (Widget);                     //Constructor
  ~Bridge ();                           //Destructor
  void  CreateDescriptionBoard();       //Create a description board
  void  ActivateBidBoard   ();          //Allow input to buttons
  void  DeActivateBidBoard ();          //Shut off input
  void  BiddingHands       ();          //Start bidding process
  void  ControlBidHands    (int, char*); //Control bidding stage
  int   DealBidding        (char*);     //Verify each bidding
  int   FindContract       (char*);     //Save a final contract
  void  PlayTricks         ();          //Start card play
  void  NewGameControl     ();          //Start a new dealer
  void  QuitWindow         ();          //Exit from window
```

```
  private:
    void  InitialData     ();              //Initialise data values
    int   BiddingIsLegal  (char *lastbid, char *btstr); //Verify bidding
    void  HandsBidDisplay (char *btstr);   //Display a player's bidding
    void  PostNextBider   (int);           //Post next bider
    int   BidOutputDisplay();              //Display a final contract
    void  FindDoubler     ();              //Find a doubler/redoubler

    //Callbacks registered with widgets in the grid
    static void BiddingCallback ( Widget, XtPointer, XtPointer );
    static void PlayCallback    ( Widget, XtPointer, XtPointer );
    static void NewGameCallback ( Widget, XtPointer, XtPointer );
    static void QuitCallback    ( Widget, XtPointer, XtPointer );
    static void BidButtonPushed ( Widget, XtPointer, XtPointer );

    Widget  Topform;            //Form Widget for the window
    BidBoard   *theBidBoard;    //An object of class BidBoard
    HandBoard  *theHands;       //An object of class HandBoard
    Command    *theCommands;    //An object of class Command
    Message    *conmessage;     //An object of class Message for comd. board
    Message    *desmessage;     //An object of class Message for des. board
    int    comeBider;           //Next bider
    int    countBider;          //Count number of bider
    int    bidtimes;            //Count number of rounds for bidding
    char   bidround[4][10];     //Save four hands of bidding
    char   bidmes[400];         //Save bidding message
    char   bidbuf[50];          //Save each round bidding
    char   BidBuf[4][10][4];    //Save 4 hands,10 round,bidding info.
    char   errmes[50];          //Save error messages
    char   lastbid[4];          //Save last bidding info.
    int    lastBider;           //Save last bider
    int    doubler;             //A doubler
    int    doubleFlag;          //Set double flag
    int    redoubleFlag;        //Set redouble flag
    int    firstBider;          //First bider
    int    passtimes;           //Count passtimes
    int    trumpSuit;           //Bidding suit
    int    finalBider;          //The last bider
    int    firstPlayer;         //First leading player
    int    flagup;              //Check no contract
    int    lastdber;            //0: not doubled;   1: doubled
    int    lastredber;          //0: not redoubled; 1: redoubled
    char   contract[4][15];     //Save bidding info.
    char   fincontract[4];      //Save a final contract
};
#endif
```